

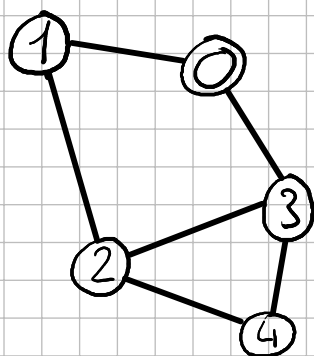
# Algoritmo di visita

Un algoritmo di visita per un grafo deve

- 1) Partire da una sorgente
- 2) Tenere traccia dei nodi scoperti
- 3) Attraversare Archi che hanno un estremo scoperto ed un estremo non ancora scoperto

## Esempio:

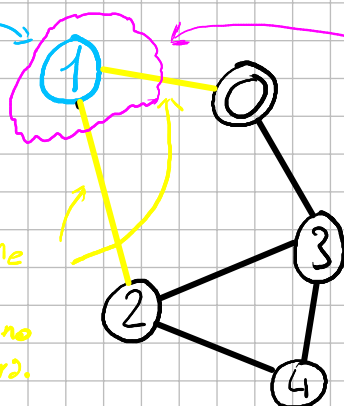
(visita non condizionata)  
visita tutto il grafo



Passo 1°) Entro nella sorgente, ovvero il primo nodo che visitiamo.

sorgente →

Archi che attraversano la frontiera.



Frontiera tra  $S$  e  $V-S$

Frontiera: Nodi Visitati

## Pseudo Codice:

```
visitaGenerica(sorg){  
  S = {sorg};  
  finché è possibile  
    scegli(u, v) tali che  $u \in S, v \notin S$   
    S = S  $\cup$  v // aggiungo ad S il nodo appena visitato.  
}
```

← dipende dall'algoritmo

## Proprietà dell'algoritmo:

Cosa deve fare  $\rightarrow$  Come lo fa

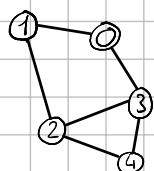
1) deve terminare  $\rightarrow$  ad ogni iterazione aggiunge un nodo a quelli scoperti, termina in  $n-1$  iterazioni

2) deve scoprire tutti i nodi raggiungibili  $\rightarrow$  mediante ad un cammino su grafo; ovvero una sequenza ordinata di nodi  $v_1, v_2, \dots, v_k$  dove

$$(v_i, v_{i+1}) \in E, \text{ con } v_{i+1} \text{ vicino di } v_i$$

significa che questo arco deve appartenere al grafo dell'arco

Es.



1, 2, 3 è un cammino ✓

1, 2, 4 non è un cammino ✗

1, 2, 1, 2 è un cammino ✓

Cammino Semplice: i nodi non si ripetono

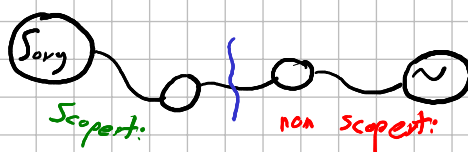
1, 2, 3 è semplice ✓

1, 2, 1, 2 non è semplice ✗

3)  $u$  raggiungibile da  $v$  se esiste in  $G$  un cammino.  $V = v_1, v_2, \dots, v_k = u$

$\rightarrow$  In questo caso dobbiamo usare una dimostrazione per assurdo, ovvero negare la tesi, nel nostro caso che  $u$  non viene scoperto

È possibile che  $u$  non venga mai scoperto?



```
visitaGenerica(sorg){  
  S = {sorg};  
  finché è possibile  
    scegli  $(u, v)$  tali che  $u \in S, v \notin S$   
     $S = S \cup v$   
}
```

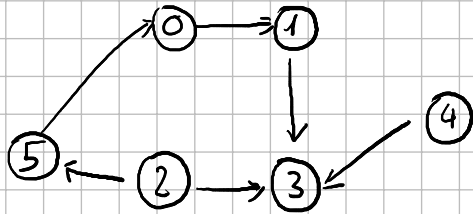
No, non è possibile che  $u$  non venga mai scoperto, perché questo pezzo di codice si occupa di questo.

Abbiamo dimostrato l'ipotesi.

## Per i grafi orientati?

L'algoritmo è ancora valido, ma non possiamo più dire che scopre sempre tutti i nodi (punto 2), perché questo dipende da come sono orientati gli archi.

Es:



Per esempio non puoi andare da 3 a 4.

Basta dire che nel punto 2 invece che usare un cammino usiamo un cammino orientato, ovvero un cammino che segue il senso della freccia.

$$N_1, N_2, \dots, N_k, (v_i, v_{i+1}) \in E$$

## Modifica dell'algoritmo di visita

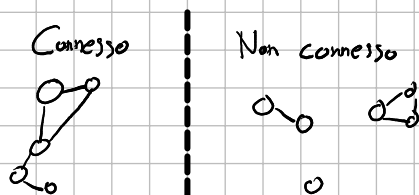
```
visitaGenerica(sorg){  
  S = {sorg};  
  • A = {} // archi attraversati:  
  finché è possibile  
    scegli(u, v) tali che  $u \in S, v \notin S$   
    S = S  $\cup$  v  
    • A = A  $\cup$  {(u, v)}  
}
```

// grafo di partenza non deve essere orientato

Con queste due aggiunte (•) siamo in grado di generare il grafo  $T(S, A)$ , un grafo connesso e aciclico.

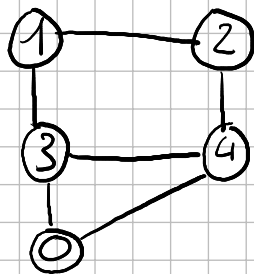
• **Connesso**: se  $\forall u, v \in V$  u è raggiungibile da v e viceversa.

Es:



• **Ciclico**: se esiste un cammino  $N_1, N_2, \dots, N_k$  con  $N_1 = N_k$ , viene anche chiamato cammino chiuso.

Es:



$1, 2, 4, 3, 1$  é un ciclo ✓

$1, 2, 4, 3$  non é un ciclo ✗

$1, 2, 4, 1$  non é un ciclo perché non esiste quel cammino ✗

Inoltre esistono:

Cicli semplici: i nodi non si ripetono

$1, 2, 4, 3, 1$  é un ciclo semplice ✓

Occhio che ciclo semplice  $\neq$  cammino semplice

Cicli non semplici: i nodi si ripetono

$1, 2, 4, 3, 0, 4, 3, 1$  non é un ciclo semplice

Cicli banali: sono cicli che ad un certo punto fanno lo stesso percorso al contrario.

Sono cicli sempre presenti, perché derivano dal fatto che il grafo é orientato

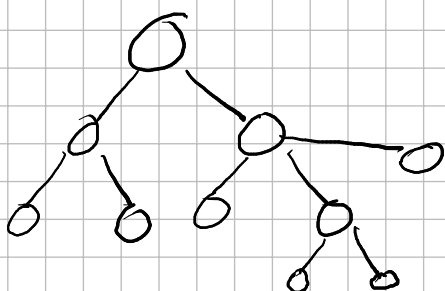
$1, 2, 1$  é un ciclo banale

$1, 2, 4, 3, 0, 4, 2, 1$  non é semplice e non é banale

Grafo aciclico: Grafo senza Cicli semplici

Adesso che abbiamo copito questi concetti possiamo andare avanti con il nostro algoritmo.

Possiamo dire che  $T$  é un albero di visita.



essendo non orientato non ha una vera radice, dato che ogni nodo può essere una radice.

Come noti non c'è la possibilità di fare

Cicli non banali.

$T$  avrà sempre  $n-1$  archi

Dimostrazione di  $T(S, A)$  è un albero di ricerca:

Dimostrazione per induzione:

Induzione su  $|S|$  ovvero i nodi scoperti

Caso Base:

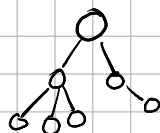
$$S = \{\text{sorg}\} \quad |S| = 1$$



Passo Induttivo:

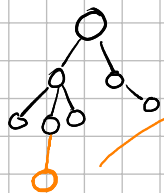
Se è vero per  $|S|=k$  è vero anche per  $|S|=k+1$

Significa che quando sono stati visitati  $k$  nodi allora la struttura costruita è effettivamente un albero



Ora facciamo finta di fare un ulteriore passo di visita, ovvero

```
visitaGenerica(sorg){  
  S = {sorg};  
  A =  $\emptyset$   
  finché è possibile  
    scegli  $(u, v)$  tali che  $u \in S, v \notin S$   
    S =  $S \cup v$   
    A =  $A \cup \{(u, v)\}$   
}
```



Il nuovo nodo viene aggiunto alla struttura già esistente, quindi la struttura rimane connessa.

In più non si creano cicli perché, per aggiungere un ciclo dovresti aggiungere un arco tra

nodi che sono già stati scoperti, ma questo

non può mai accadere perché l'arco viene

collegato con un nodo che non era ancora

scoperto.  $v \notin S$

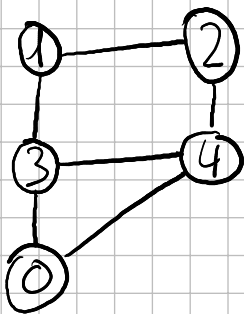
In questo modo abbiamo dimostrato l'ipotesi induttiva.

## Visita BFS (Ricerca in ampiezza):

- la frontiera viene gestita con una coda.

```
visitaBFS(sorg){  
  S = {sorg};  
  A =  $\emptyset$   
  coda  $\leftarrow$  sorg  
  finché coda non vuota  
    u  $\leftarrow$  coda // dequeue / rimozione da coda  
    per ogni vicino u di v  
      se v  $\notin$  S // se non è stato scoperto  
        S = S  $\cup$  v  
        coda  $\leftarrow$  u  
        A = A  $\cup$  {(u,v)}  
}
```

## Esempio Funzionamento



prendiamo come sorgente il nodo 1

Coda

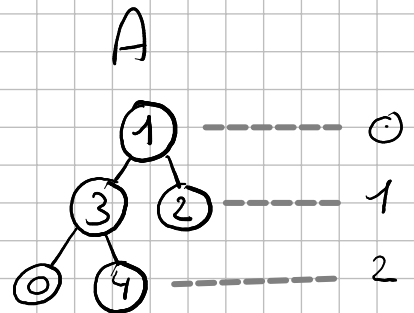
~~1~~ ~~3~~ ~~2~~ ~~0~~ ~~4~~

S

1, 3, 2, 0, 4

A

(1,3) (1,2) (3,0) (3,4)



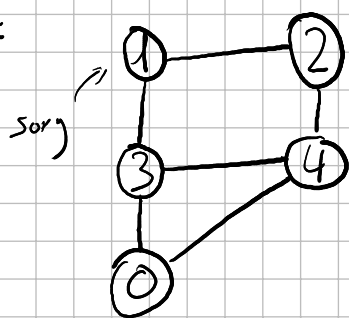
A ti dice la distanza dei nodi dalla sorgente

Distanza: lunghezza

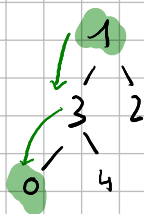
minima per arrivare ad un nodo.

Utilizzando l'albero di ricerca che b visita BFS ci ha restituito possiamo vedere qual è il percorso più veloce per andare da un nodo all'altro.

Es:



Voglio andare da 1 a 0, qual è il percorso migliore?



Come vedi: fornisce il percorso più breve, ovvero 1,3,0 e non per esempio 1,2,4,3,0

## Visita DFS (in profondità)

È molto simile alla BFS, semplicemente sostituiamo la coda con una pila. Detto meglio gestiamo la frontiera con una pila.

```
visitaDFS(sorg){  
  S = {sorg};  
  A = ∅  
  pila ← sorg  
  finché pila non vuota  
    u ← pila // POP  
    per ogni vicino u di v  
      se v ∉ S // se non è stato scoperto  
        S = S ∪ v  
        pila ← u  
        A = A ∪ {(u,v)}  
}
```

Tutte e 3 le proprietà sono verificate

Ma abbiamo davvero bisogno di una pila esplicita?

No! Possiamo sfruttare la ricorsione.

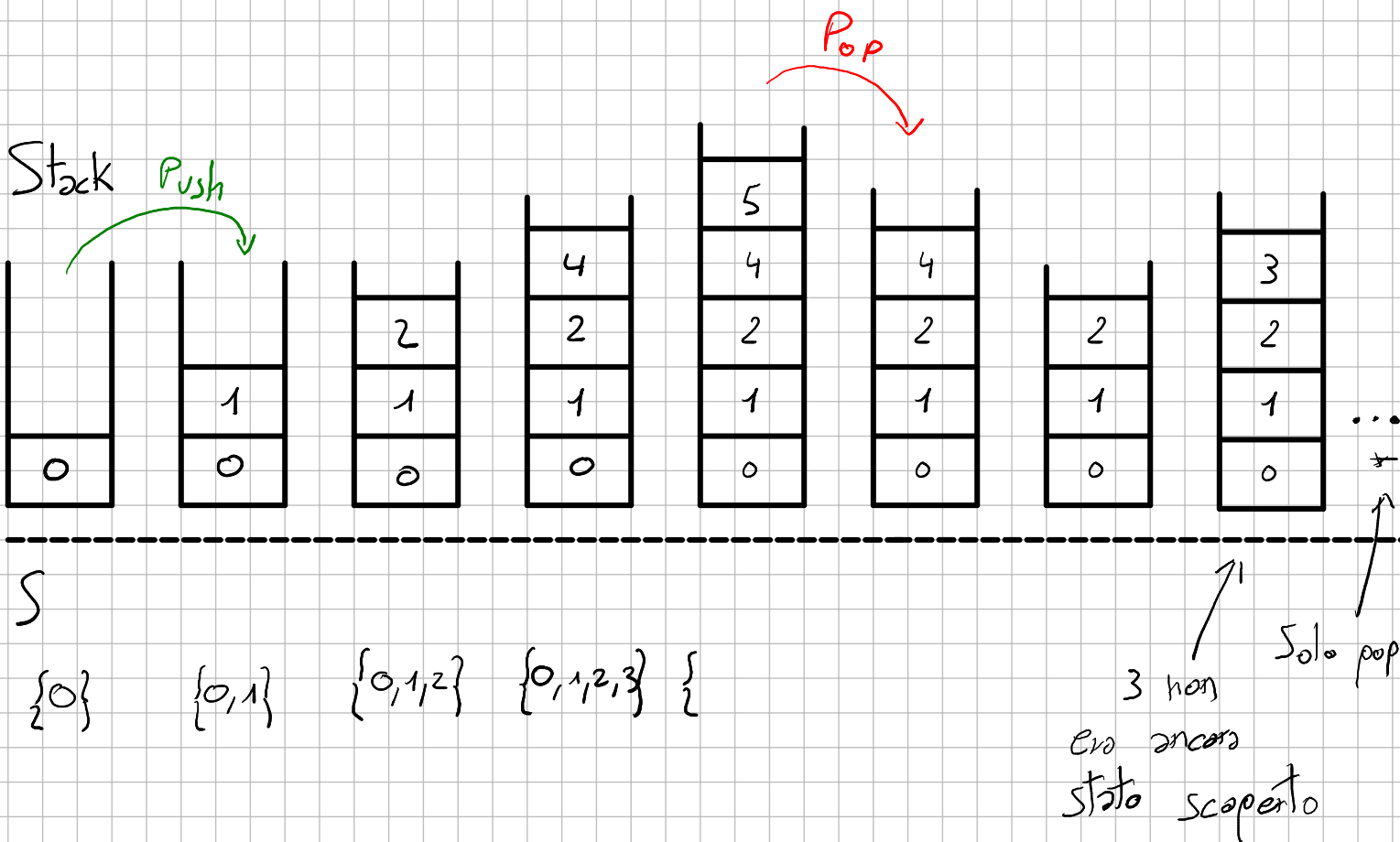
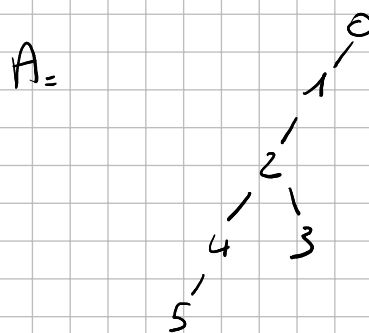
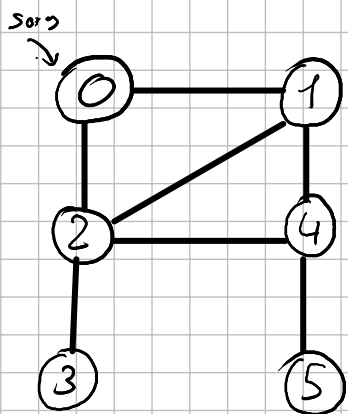
$S = \emptyset$   
 $A = \emptyset$

variabili globali

```

visitaDFS(sorg){
    S = S ∪ {sorg}
    per ogni vicino v di sorg
        se v ∉ S
            A = A ∪ {(sorg, v)}
            visitaDFS(v) // caso ricorsione
}
    
```

Esempio Funzionamento:





Nella visita DFS l'albero che ci viene restituito non può essere usato per sapere la distanza tra due nodi

Tutto molto bello, ma quanto costano gli algoritmi BFS e DFS?

```
visitaBFS(sorg){  
  S = {sorg};  
  A = ∅  
  coda ← sorg  
  finché coda non vuota  
    u ← coda  
    per ogni vicino v di u  
      se v ∉ S  
        S = S ∪ v  
        coda ← u  
        A = A ∪ {(u,v)}  
}
```

```
S = ∅  
A = ∅  
  
visitaDFS(sorg){  
  S = S ∪ {sorg}  
  per ogni vicino v di sorg  
    se v ∉ S  
      A = A ∪ {(sorg,v)}  
      visitaDFS(v)  
}
```

È palese che entrambi gli algoritmi abbiano costo

$O(n)$ , perché essendo algoritmi di visita scorrono tutto il grafo.

In particolare abbiamo  $n-1$  cicli esterni (finché coda non vuota), ma non tutti i cicli sono uguali, perché magari un nodo ha tantissimi vicini.

• In un G. orientato abbiamo:  $\sum_{out} (n_1) + \sum_{out} (n_2) + \dots + \sum_{out} (n_n) = m$

↖ numero archi

• In un G. non orientato abbiamo:  $\sum (n_1) + \sum (n_2) + \dots + \sum (n_n) = 2m = O(m)$

↖ perché ogni nodo ha sia un arco entrante che uno uscente?

Ma i vicini si trovano effettivamente così facilmente?

No, perché siamo dando per scontato che trovare i vicini di un nodo abbia costo unitario, ma non è così!

Dipende come rappresentiamo il grafo, se con una lista di adiacenza o con una matrice di adiacenza.

Con la lista trova facilmente i vicini perché basta scorrere la lista, invece nella Matrice il costo aumenta perché devi scorrere righe e colonne.

Quindi abbiamo che

- Lista di adiacenza:  $O(n+m)$

- Matrice di adiacenza  $O(n+m^2)$